Data Structure Content

Intro

Any program consists of **code + data.**
RAM consists of **Heap, stack, global variables, code**.

- **Heap**: This is the area of memory used for dynamic memory allocation. Variables allocated here persist until they are explicitly deallocated (using `delete` in C++).
- **Stack**: The stack is used for function calls and local variables. Each time a function is called, a new stack frame is created, which holds function parameters, return addresses, and local variables. The stack automatically cleans up when the function returns.
- **Global/Static Variables**: This section holds variables that are declared outside of functions, such as global variables and static variables. These variables have a lifetime that spans the entire duration of the program.
- **Code (Text Segment)**: This section contains the actual machine code that the CPU executes, typically read-only.

What is a data strucure?

A data structure is a scheme (مخطط) for organizing data in the memory of a computer for efficient storage and efficient manipulation.

Time Complexity

Time complexity refers to the amount of time an algorithm takes to complete as a function of the size of its input.

يعني مقدار الوقت اللي كود معين بيحتاجه عشان يتنفذ والوقت ده مبني على حجم الداتا أو الانبوت اللي داخله

It helps to estimate the performance of an algorithm, especially when working with large inputs. Time complexity is usually expressed using Big-O notation, which describes the *upper bound or worst case* of an algorithm's runtime as the input size grows.

Types of complexity

1. Best Case or lowerbound **Ω (Omega)**
2. Average case or upper & lower bound **θ (Theta)**.

Ramez Medhat

3. Worst case or upper bound **O (Big-O)**.

Methods of Calculating Time Complexity

1. Frequency Count

The frequency count method involves counting how many times each statement or operation in an algorithm is executed.

Example:

Frequency count method

```
Max = A[1]
For I = 1 to N
      If (A[I] > Max)
            Max = A[I]
Return Max
```

Running Time:

Statement	# Operations	Total
`Max = A[1]`	Takes 2 steps: one for = and one for [ ]	2
`For I = 1 to N`	**Initialization :** 1 step **Increment:** N steps (till breaking loop) **Comparison:** N + 1 steps (1 init. & N incr.'s)	2N+2
`If (A[I] > Max)`	Takes 2 steps: one for > and one for [ ]	2N
`Max = A[I]`	Takes 2 steps: one for = and one for [ ]	2N (worst case)
`Return Max`	Takes 1 step	1
	Running Time T(N) =	**6N + 5**

Dr. Mahm

2. Asymptotic Analysis

it focuses on describing the behavior of an algorithm as the input size grows towards infinity. It eliminates constant factors and lower-order terms, providing a general classification of the algorithm's efficiency.

The Frequency count method gives you the **Exact time**, but we care more about the **Order or Asymptomatic time الوقت التقريبي**, so we must follow these rules:

1. *Statements:* Order of 1
2. *Conditions:* choose the max
3. *Loops:* Order of body $\times$ number of iterations
4. *constants are removed.* $O(6n + n^2 + 5) = O(n + n^2)$
5. *the biggest order is the dominant* $O(n + n^2) = O(n^2)$

Common Orders

- **O(1)**: Constant time, the algorithm takes the same time regardless of input size.

```
int n = 5;
```

- **O(log n)**: Logarithmic time, the algorithm's time grows logarithmically as the input size increases. (كل مره بقلل عدد العمليات للنص)

```
for(int i = 0; i < n; i *= 2) {}
```

- **O(n)**: Linear time, the time increases proportionally with the input size.

```
for(int i = 0; i < n; i++) {}
```

- **O(n log n)**: Quasi-linear time, common in algorithms like merge sort and quicksort.

```
for(int i = 0; i < n; i++) {
        for(int i = 0; i < n; i *= 2) {}
}
```

- **O(n²)**: Quadratic time, time grows with the square of the input size, often found in algorithms that use nested loops.

```
for(int i = 0; i < n; i++) {
        for(int i = 0; i < n; i++) {}
}
```

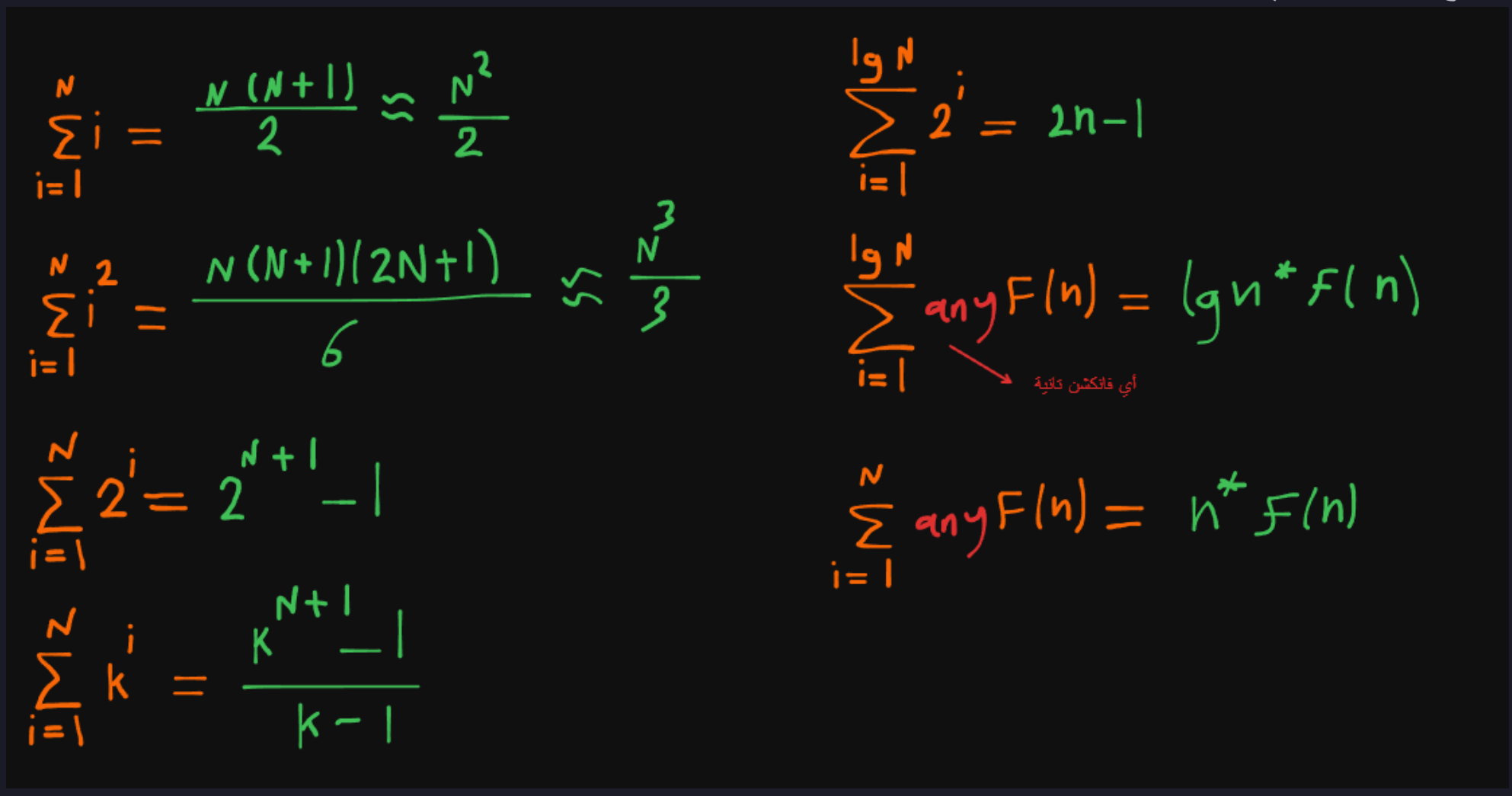
- **O(2^n)**: Exponential time, the time doubles with each additional input size.

```
int fibonacci(int n) {
        if (n ⩽ 1) return n; // Base case
        return fibonacci(n - 1) + fibonacci(n - 2); // Recursively call fibonacci
}
```

- **O(n!)**: Factorial time, very inefficient, often found in problems that try all possible permutations.

Using Equations

بصمج المعادلات دي


$\sum_{i=1}^{N} i = \frac{N(N+1)}{2} \approx \frac{N^2}{2}$
$\sum_{i=1}^{N} i^2 = \frac{N(N+1)(2N+1)}{6} \approx \frac{N^3}{3}$
$\sum_{i=1}^{N} 2^i = 2^{N+1} - 1$
$\sum_{i=1}^{N} k^i = \frac{k^{N+1} - 1}{k - 1}$
$\sum_{i=1}^{\lg N} 2^i = 2n - 1$
$\sum_{i=1}^{\lg N} anyF(n) = \lg n * F(n)$
أي فانكشن تانية
$\sum_{i=1}^{N} anyF(n) = n * F(n)$

Data structures types

1. Primitive Data Structure

These are the basic data types provided by the programming language

- Integer
- Float/Double
- Character
- Boolean

2. Non-Primitive Data Structures

These are more complex data structures derived from primitive data types.

A. Linear Data Structures

Elements are stored in a sequential order.

Direct Access

- **Array**:
  - A collection of elements of the same data type stored at contiguous memory locations.

Sequential access

- **Linked List**:
  - A sequence of elements where each element points to the next.
  - Types:
    - *Singly Linked List*: Each node points to the next node.
    - *Doubly Linked List*: Each node points to both the previous and the next node.
    - *Circular Linked List*: Last node points to the first one.
- **Stack**:
  - A collection based on LIFO (Last In, First Out).
- **Queue**:
  - A collection based on FIFO (First In, First Out).

b. Non-Linear Data Structures

Elements are not stored sequentially, and there may be multiple relationships between elements.

Hierachical

Data is organized in a hierarchy, forming a parent-child relationship. Each element (node) has a specific position in the hierarchy.

- **Tree**:
  - A hierarchical structure with a root node and children.
  - Types:
    - **Binary Tree**
    - **Binary Search Tree (BST)**
- **Heap**

Unordered

Data is stored without any specific order or hierarchy, and there are no defined relationships between elements.

- **Graph**
- **Sets**
- **Hash-Tables**

Other types of data structures

1. **Static:** its size is fixed ثابت
2. **Dynamic:** its size is not fixed and can be modified during runtime.
3. **Homogeneous:** all the elements are the same type (array).

4. **Non-Homogenous:** may or may not be the same type (structs).

Physical data strucutre VS logical data structure

- **Physical:** doesn't depend internally on any other data structure (static array), represent the raw storage structures that directly manage memory.
- **Logical:** depends on other data structures, focuses on the logic of operations done on data (stack depends on arrays or linked list)

Abstract Data type

defines a data type in terms of **a type** and **a set of operations** on that type
It focuses on what the structure does and ignoreing how it does its job.
Doesn't specifiy how the the data type is implemented
Data structures is the physical implementation of an ADT.
بختصار هو نص بيوصل الداتا ستراكشر ده عباره عن ايه وايه الوظايف اللي اقدر اعملها بيه

Stack ADT

A Stack contains elements of same type arranged in sequential order. All operations takes place at a single end that is top of the stack and following operations can be performed:

push() – Insert an element at one end of the stack called top.

pop() – Remove and return the element at the top of the stack, if it is not empty.

peek() – Return the element at the top of the stack without removing it, if the stack is not empty.

size() – Return the number of elements in the stack.

isEmpty() – Return true if the stack is empty, otherwise return false.

isFull() – Return true if the stack is full, otherwise return false.

Static Array

it is a basic physical data structure used to store a fixed-size collection of elements of the same type, these elements are arranged in contingous memory locations, allowing each element to be accessed directly using an integer index.

Static array characteristics

- ***Fixed Size:*** Once an array is declared, its size cannot be changed during runtime. You need to know the number of elements you will store in the array beforehand. Memory is allocated on the **stack**. The allocation is done at **compile time**, and the size is fixed.
- ***Homogeneous Elements:*** All elements in an array must be of the same data type, such as all integers, all floats, or all characters.
- ***Indexed by Integers***: Each element in an array is assigned a unique integer called an index, which identifies its position within the array. Indexing usually starts at 0.
- ***Efficient Access:*** Because of the way arrays are stored in memory (contiguously), accessing any element by its index is very efficient. This direct access using the index is often referred to as "random access."

Types of arrays

- **1 Dimentional array:** simplest type, single line of elements, uses 1 index.
- **2 Dimentional array:** matrix, consist of rows and columns, accessed by 2 indecies.
- **Multi-Dimentional array:** more than 2D, used in senarios that requires complex data representations.

Declaration of an array

```
// 1d array
datatype name[size] = {elements, ... , elements};
int a[5] = {1,2,3,4,5};

// 2d array
int name[columns][rows] = {
        {1,2,3},
        {1,2,3},
        {1,2,3},
}
```

Array ADT

Array ADT	
Array is a data structure that is used to store multiple values of similar data types in a contiguous memory location using the following operations.	
Append	Add an element at the end
Insert	Insert an element at specific location
Remove	Remove an element from the end
Delete	Delete an element from specific location
Get	Access/Return an element at specific index
GetSize	Return the current size of the array
Reverse	Reverse array elements
Search	Look up an element on array, return its index
Display	Display array elements
Sort	Sort array elements in ascending and descending order

Based on this ADT we can implement it using class.

Encapsulate all data items and functions (operations) in a class

Then, we can use this class as a data structure

Let's see how it done

Class

```
template<typename t>
class StaticArray {
        private:
            int size;
            int capacity;
            t a[];

        public:
                // Paramterized Constructor
            StaticArray(int capacity) : capacity(capacity), size(0) {
                a[capacity];
            }

                // Copy Constructor
            StaticArray(const StaticArray<t> &b) {
                a[b.capacity];
                size = b.size;
                capacity = b.capacity;
                for (int i = 0; i < size; ++i) {
                    a[i] = b[i];
                }
            }

            ~StaticArray(){
                    // empty because data is deleted automaticlly from stack
            }
}
```

Append

```
void append(t element) {
        // Check if array is full
        if(size == capacity) {
                throw std::out_of_range("ERROR: Array is full");
        }

        a[size] = element;
        size++;
}
```

Insert

```
void insert(t element, int index) {
        // Check if array is full
        if(size == capacity) {
                throw std::out_of_range("ERROR: Array is full");
        }
        // Check if index is out of range
        if(index > size) {
                throw std::out_of_range("ERROR: Invalid index");
        }

        // shift all elements after (index) to the right
        for(int i = size-1; i ⩾ index; i--) {
                a[i+1] = a[i];
        }

        a[index] = element;
        size++;
}
```

Delete End

بنعملها كده عشان لما نييجي نضيف عنصر جديد هنضيفه عند ال size فا مش هيفرق

```
void delete_end() {
        size--;
}
```

Delete at Index

لما نحرك كل العناصر اللي بعد العنصر اللي عايزين نمسحه للشمال فبالتالي العنصر ده هيتكتب فوقه القيمة اللي بعده

```
void delete_at(int index) {
        // shift all elements after (index) to the left
        for(int i = index; i < size-1; i++) {
                a[i] = a[i+1];
        }
```

```
        size--;
}
```

Get element, size

```
t getElement(int index) {
        return a[index];
}

int getSize() const {
        return size;
}
```

Reverse

```
// Helper function
private:
void swap(t &x, t &y) {
        t temp = x;
        x = y;
        y = temp;
}

public:
void reverse() {
        int l = 0, r = size - 1;
        while(l < r) {
                swap(a[l], a[r]);
                l++; r--;
        }
}
```

Linear search

```
// O(n), slow but require no sorting for the array
int linearSearch(t element){
        for(int i = 0; i < size; i++){
                if(a[i] == element){
                        return i;
                }
        }
        return -1;
}
```

Binary search

```
// O(LogN), fast but require sorting the array first
int binary_search(t element) {
```

```
        int l = 0, r = size - 1;
        while (l <= r) {
                int mid = l + (r - l) / 2;
                if (a[mid] == element) {
                        return mid;
                } else if (a[mid] < element) {
                        l = mid + 1;
                } else {
                        r = mid - 1;
                }
        }
        return -1;  // Return -1 if element is not found
}
```

Display

```
void displayArray() {
        cout << "[ ";
        for (int i = 0; i < size; i++) {
                cout << a[i] << ", ";
        }
        cout << "]" << endl;
}
```

Dynamic Array

It is similar to a static array, but its size is modifiable during program runtime, dynamic array elements occupy a contigous block of memory.

once a static array has been created, it is allocated a predetermined amount of memeory and its size **cannot be changed**, but dynamic array **can expand its size even after it has been filled** as it grows its memory size by a certian factor when there is a need (also more potential for errors).

Dynamic arrays are allocated in heap.

To allocate:

```
int *p = new int[500];
```

this line allocates a segment in memory of type array of int and of size `500`

To initilize it to zero add `()` at the end

```
int *p = new int[500]();
```

Grabage problem

- The memory allocated for array in memory must be manually released, or else the program will have **memory leak**
- use `delete` keyword

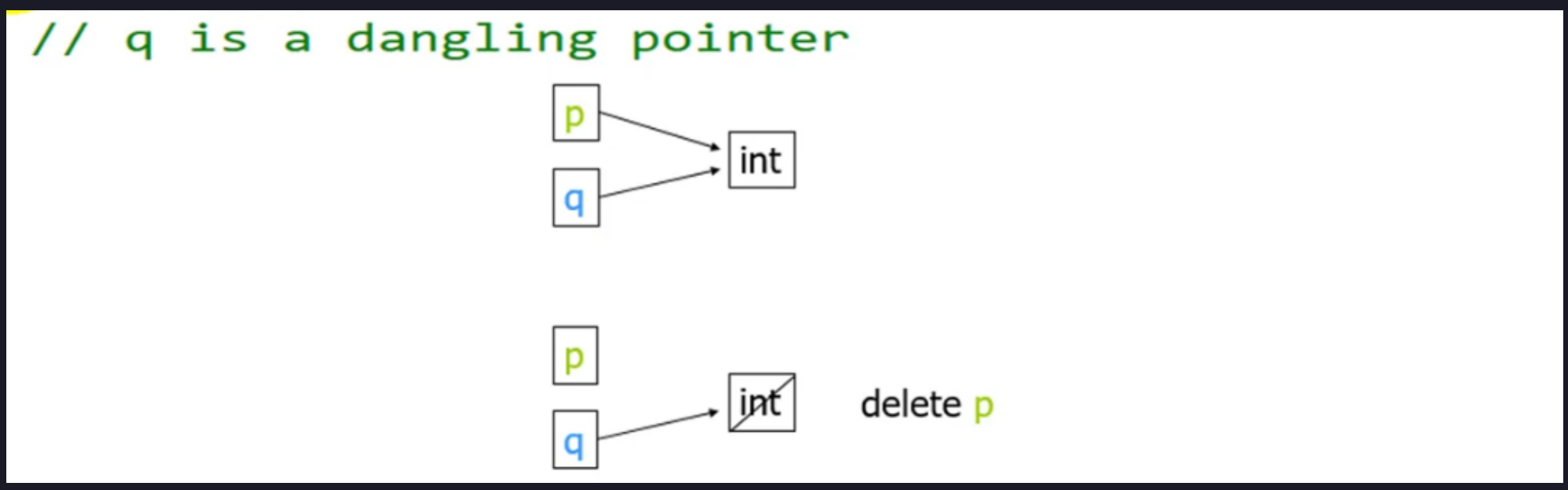
```
delete []p;
```

accesing array element

```
p[0] = *p; // first element
p[1] = *(p+1); // second element
p[2] = *(p+2); // third element and so on
```

Dangling pointer problem

It happens when two pointers `p` and `q` both points at some data, then you delete the data using one of them and try to access it using the other pointer.



Implementation:

The implementation of dynamic array is the same as static array, but we will change few things:

Class

```
template<typename t>
class DynamicArray {
        private:
            int size;
            int capacity;
            t *a; // this became a pointer

        public:
            DynamicArray(int capacity = 2) : capacity(capacity), size(0) {
                a = new t[capacity]; // using new keyword
```

```
    }

    DynamicArray(const DynamicArray<t> &b) {
        size = b.size;
        capacity = b.capacity;
                a = new int[capacity];
        for (int i = 0; i < size; ++i) {
            a[i] = b[i];
        }
    }

    ~DynamicArray() {
        delete []a; // deleting data
    };

        // copy assignment operator (deep copy)
    DynamicArray& operator=(const DynamicArray &other){
                if(this != &other) { // check self-assignment
                    delete []arr; // delete old data
                    // copy data from other
                    size = other.size;
                    arr = new int[size];
                    for (int i = 0; i < size; ++i) {
                        arr[i] = other.arr[i];
                    }
                }
            }
};
```

Resizing the dynamic array

we need this function to be able to edit the size of array at any time during runtime.

```
void resize(int newCapacity) {
        // 1. Allocating new memory location with the new cap
    t *b = new t[newCapacity];
    // 2. copying the old array elements into the new one
    for (int i = 0; i < size; i++) {
        b[i] = a[i];
    }
    // 3. freeing old space from memory
    delete[] a;
        // assigning the old pointer to the new memory location
    a = b;
    // updating the new capacity
    capacity = newCapacity;
}
```

Adding elements

Ramez Medhat

before each time we add elements we check if the array is full, if yes we double the size of the array.

```
if(size == capacity) {
        resize(2 * capacity);
}
```

Deleting elements

after each time we delete elements we check if the array is less than or equal 25% full, if yes we shrink the size to the half.

```
if(size <= capacity/4) {
        resize(capacity/2);
}
```

Notes on Arrays

- ***RAII (Resource Acquisition Is Initialization)*** is a design principle in C++ where resources like dynamic memory are acquired and released automatically through constructors and destructors. Ensure that the constructor allocates the dynamic array and the destructor deallocates it.
- In C++, if a class manages a resource like dynamic memory, you need to properly implement ***the Rule of Three***:
  - **Destructor**: To free the memory.
  - **Copy Constructor**: To correctly copy the array when copying objects.
  - **Copy Assignment Operator**: To correctly handle assignments between objects.
- Since the size of dynamically allocated arrays isn't known at compile time, ***you can't use range based loops directly.*** However, you can still use a traditional for loop with auto to simplify the type handling.
- `copy()` **Function**
  - it's in the header `#include <algorithm>`
  - it takes 3 parameters `copy(start, end, destination)`

```
// for resize
std::copy(arr , arr + size , new_array);
// for insertion
std::copy_backward(arr + index, arr + size , arr + index + 1);
// for deletion
std::copy(arr + index + 1, arr + size, arr + index);
```

- To make your class behave like a native array, you can overload the `operator[]` This allows you to access elements in the same way you

access elements in a built-in array

```
t& operator[](int index) {
    if (index ⩾ size || index < 0) {
        throw std::out_of_range("Array index out of range");
    }
    return a[index];
}
```

```
DynArray Obj;
cout << Obj.get(5);
cout << Obj[5];
```

Linked List

A **linked list** is a linear physical data structure in which elements, called **nodes**, are stored in a sequence. Each node contains two parts:

1. **Data**: The value or information stored in the node.
2. **Pointer (Next)**: A pointer (or reference) to the next node in the sequence.

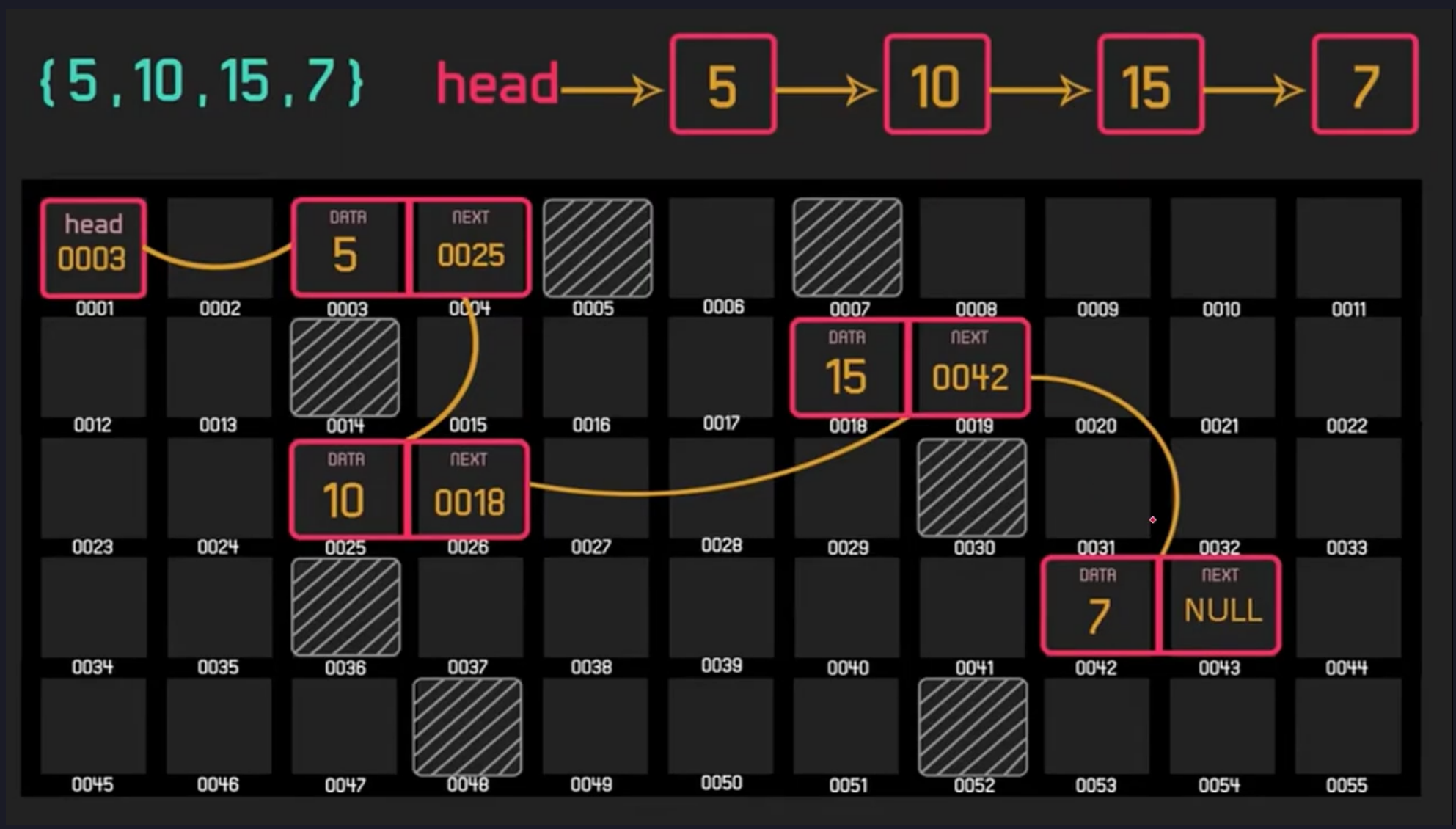
The key characteristic of a linked list is that the elements (nodes) are not stored in contiguous memory locations like arrays. Instead, each node points to the next one, allowing for dynamic memory allocation and flexible size.

Head Pointer:

- The **head** is a pointer or reference to the first node in a linked list.
- It acts as the starting point for accessing or traversing the entire list.

NULL pointer

- It is used as a marker to signify the end of the list (Tail). The `next` pointer of the last node in a linked list points to **NULL**, indicating there are no further nodes in the sequence.



Self Referential Structure

Strucutures in which one or more pointers points to the structure of the same type.

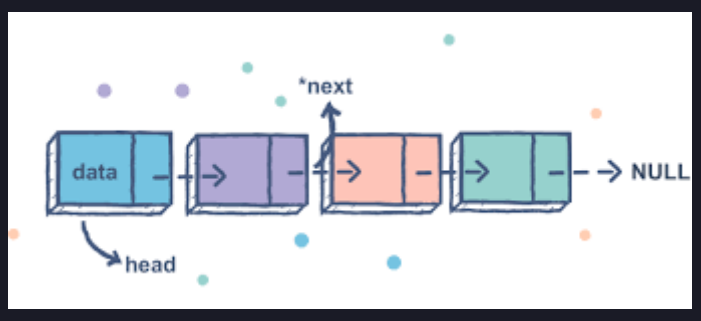
```
struct Node {
        int data;
        Node *next; // points at a structure of the same type

public:
        Node(int val) : data(val), next(nullptr) {}
}
```

these are used in linked list to make every node points at the next node.

Types of Linked List

1. Single linked list

- Each node has a single pointer to the next node.
- It can only be traversed in one direction (from head to tail).
- The last node's pointer is `null`, indicating the end of the list



Class

```
template <class t>
class SingleLL {
    private:
        Node<t> *head;
        int size;

    public:
                SingleLL() : head(nullptr), size(0) {}

                ~SingleLL(){
                        // Defined below
            clear();
        }
};
```

Traverse

```
void traverse() const {
        Node<t> *current = head;
        // until not reached the last node
        while (current != nullptr) {
                cout << current->data << " ";
                // move pointer to the next node
                current = current->next;
        }
        cout << endl;
}
```

Insert at begin

```
void insert_at_begin(t val) {
        Node<t> *newNode = new Node<t>(val);
        newNode->next = head;
        head = newNode;
        size++;
}
```

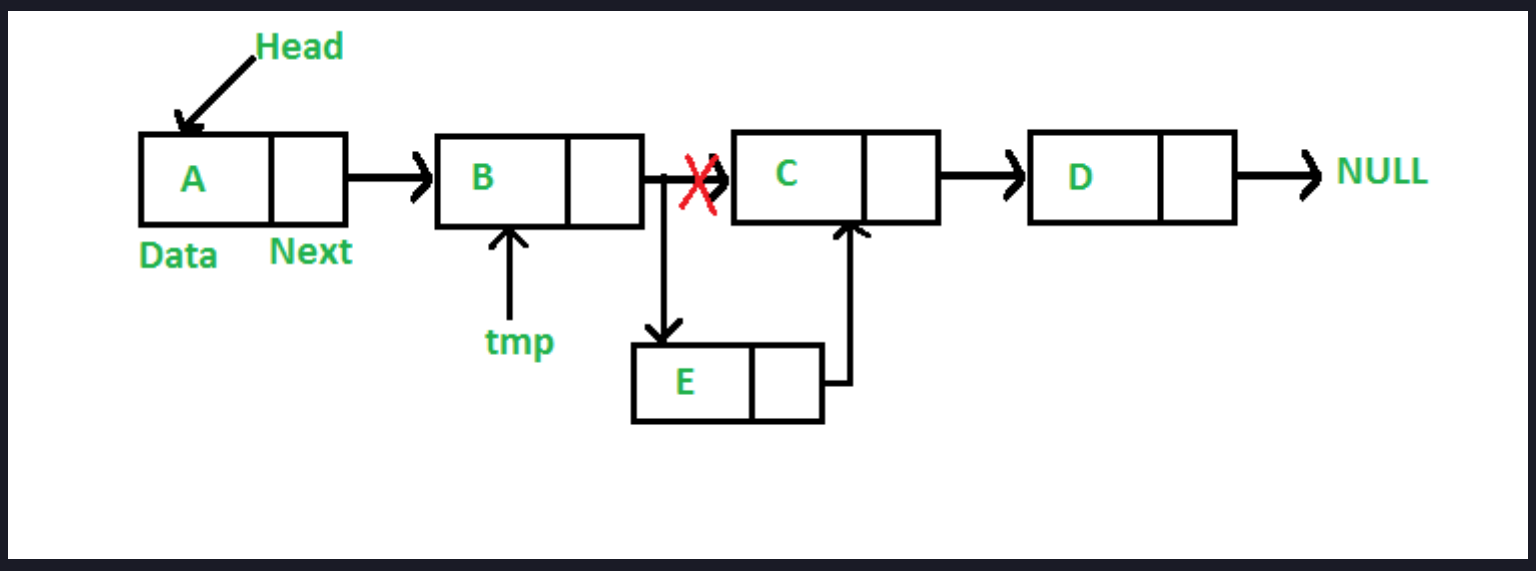
Insert at end

```
void insert_at_end(t val) {
        // if the linked list is empty, add node to the head.
        if(head == nullptr) {
                head = new Node<t>(val);
                return;
        }

        // Go to the last node
```

```
        Node<t> *temp = head;
        while (temp->next != nullptr) {
                temp = temp->next;
        }

        // make it point at the new node
        temp->next = new Node<t>(val);

        size++;
}
```

Insert at

```
void insert_at(t val, int index) {
        if(index < 0 || index >= size) {
                throw out_of_range("index out of range");
        }

        if(index == 0) {
                insert_at_begin(val);
                return;
        }

        Node<t>* newNode = new Node<t>(val);
        Node<t>* temp = head;
        for (int i = 1; i < index; i++) {
                temp = temp->next;
        }
        newNode->next = temp->next;
        temp->next = newNode;

        size++;
}
```



Delete at begin

```
void delete_at_begin() {
        if(head == nullptr) {
                cout << "Empty list" << endl;
                return;
```

```
        }
        Node<t> *temp = head;
        head = head->next;
        delete temp;
        size--;
}
```

Delete at end

```
void delete_at_end() {
        // empty
        if(head == nullptr) {
                cout << "Empty list" << endl;
                return;
        }

        // has one element
        if(head->next == nullptr) {
                delete head;
                head = nullptr;
                size--;
                return;
        }

        // more than one element
        Node<t> *temp = head;
        while (temp->next->next != nullptr) {
                temp = temp->next;
        }

        delete temp->next;
        temp->next = nullptr;
        size--;
}
```

Delete at

```
void delete_at(int index) {
        if(index < 0 || index >= size) {
                throw out_of_range("index out of range");
        }

        if(index == 0) {
                delete_at_begin();
        } else if(index == size - 1) {
                delete_at_end();
        } else {
                Node<t>* temp = head;
                for (int i = 1; i < index; i++) {
                        temp = temp->next;
```

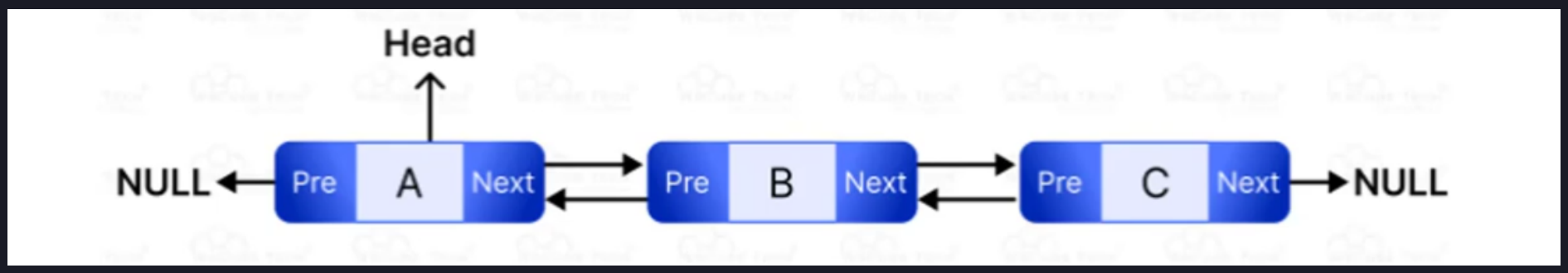
```
                }
                Node<t> *Nt = temp->next;
                temp->next = Nt->next;
                delete Nt;

                size--;
        }
}
```

Clear

```
void clear() {
        Node<t>* current = head;
        while (current != nullptr) {
                Node<t> *temp = current->next;
                delete current;
                current = temp;
        }
        head = nullptr;
        size = 0;
}
```

2. Doubly linked list

- Each node has two pointers: one to the next node and one to the previous node.
- It can be traversed both forwards and backwards



It has a very similar implmentation to a single linked list but differ in small things

Node

```
struct Node {
    T data;
    Node* next;
    Node* prev; // prev pointer

    Node(T val) : data(val), next(nullptr), prev(nullptr) {}
};
```

Insert at begin

```
void insert_at_begin(T val) {
        Node<T>* newNode = new Node<T>(val);
        if (head == nullptr) {
                head = tail = newNode; // tail is added
        } else {
                newNode->next = head;
                head->prev = newNode; // this is also added
                head = newNode;
        }
        size++;
}
```

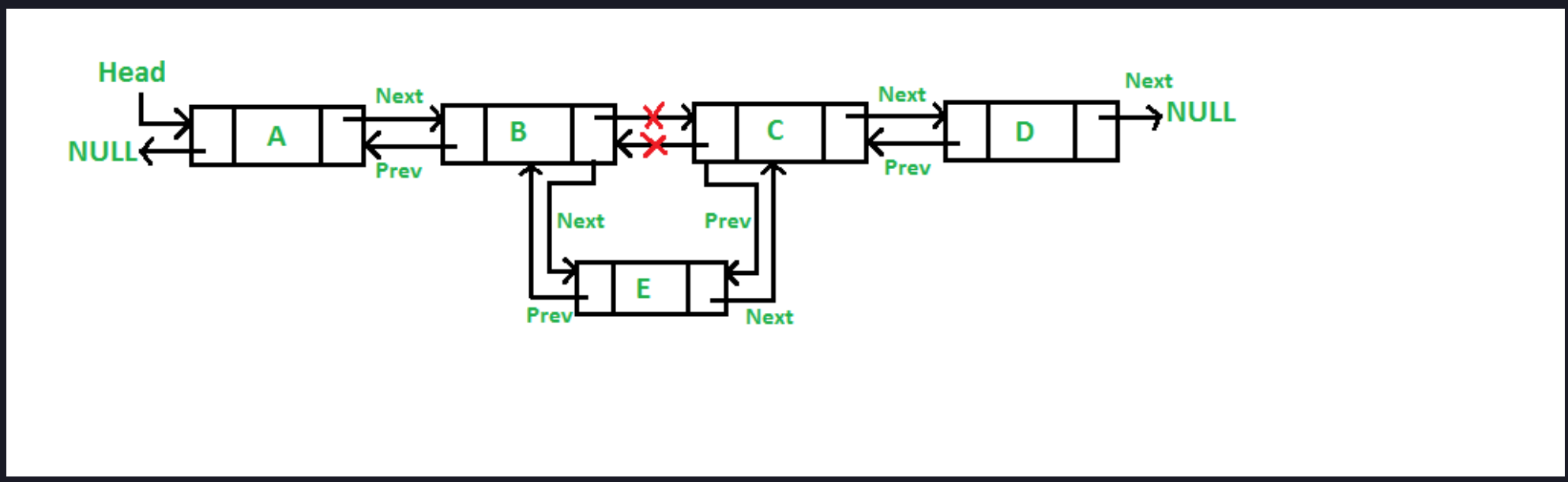
Insert at end

```
void insert_at_end(T val) {
        Node<T>* newNode = new Node<T>(val);
        if (tail == nullptr) {
                head = tail = newNode; // tail is added
        } else {
                newNode->prev = tail;
                // we don't need to loop to reach the end, we already have a tail
                tail->next = newNode;
                tail = newNode;
        }
        size++;
}
```

Insert at

```
void insert_at(T val, int index) {
        if (index > size || index < 0) {
                throw out_of_range("Index out of range");
        }

        if (index == 0) {
                insert_at_begin(val);
        } else if (index == size) {
                insert_at_end(val);
        } else {
                Node<T>* newNode = new Node<T>(val);
                Node<T>* current = head;
                for (int i = 0; i < index; i++) {
                        current = current->next;
                }
                newNode->next = current;
                newNode->prev = current->prev;
                current->prev->next = newNode;
                current->prev = newNode;

                size++;
```

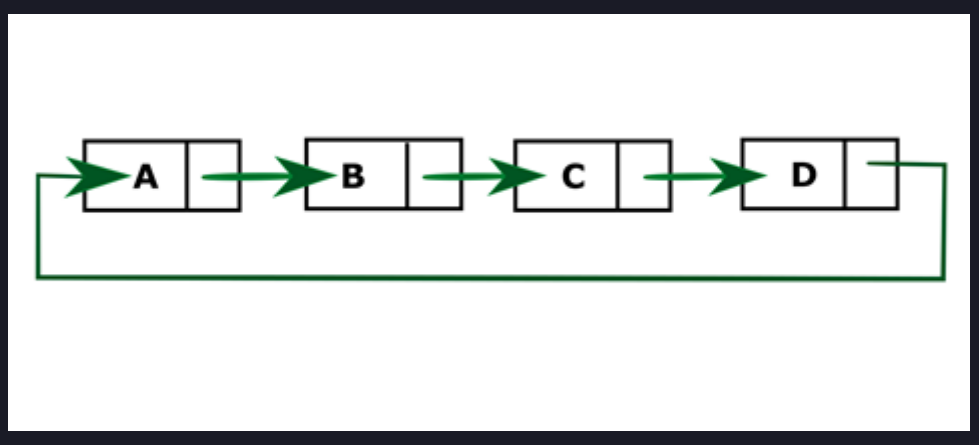
```
        }
}
```



And the rest of the deteting function is the same, we put into consideration that there's a tail that must be updated with a new value.

3. Circular linked list

- The last node points back to the first node instead of `null`, creating a loop.
- Can be singly or doubly linked



وانت بتلف مش هتوصل ل `null` لا انت هتوصل للhead تاني، الدكتور مشرحهاش ولكن ممكن تفكر فيها مع نفسك (:

Stack

It is a linear-squential-logical data structure, It follows the **Last In, First Out (LIFO)** principle. This means that the most recently added element is the first one to be removed.

Stack are used in **function calls** for almost all programming evnironments.

Stack operations

1. ***Push***: adds an element to the top of the stack, this element is the most accessible and will be the first to be removed.

2. ***Pop***: Removes the top element from the stack, which was the item that was most recently added.
3. ***Peek/Top***: retrives the top element without removing it. (بترجع أخر قيمة اتضافت)

Implementation

Stack can be built with both dynamic array or linked list.
الاسهل تستخدم `linked list` كل اللي عليك فال `push` تضيف عنصر فالاول وفال `pop` تمسح عنصر من *الاول* وفال `peek` هترجع العنصر اللي عند ال *head* (اللي *فالاول*)

Push

```
void push(t val){
    // allocation
    Node<t> *newNode = new Node<t>(val);

    if(head == nullptr) {
        head = newNode;
    } else {
        newNode->next = head;
        head = newNode;
    }
}
```

Pop

```
void pop(){
    if(head == nullptr){
        cout << "Stack is empty!\n";
    } else {
        Node<t> *temp = head;
        head = head->next;
        delete temp;
    }
}
```

top/peak

```
void top(){
    if(head == nullptr){
        cout << "Stack is empty!\n";
    } else {
        cout << head->data << "\n";
    }
}
```

ولو عملته باستخدام دايناميك ارراي برضو هتضيف عند ال size وتمسح من عند ال size (اللي هوا من عند النهاية يعني)

Queue

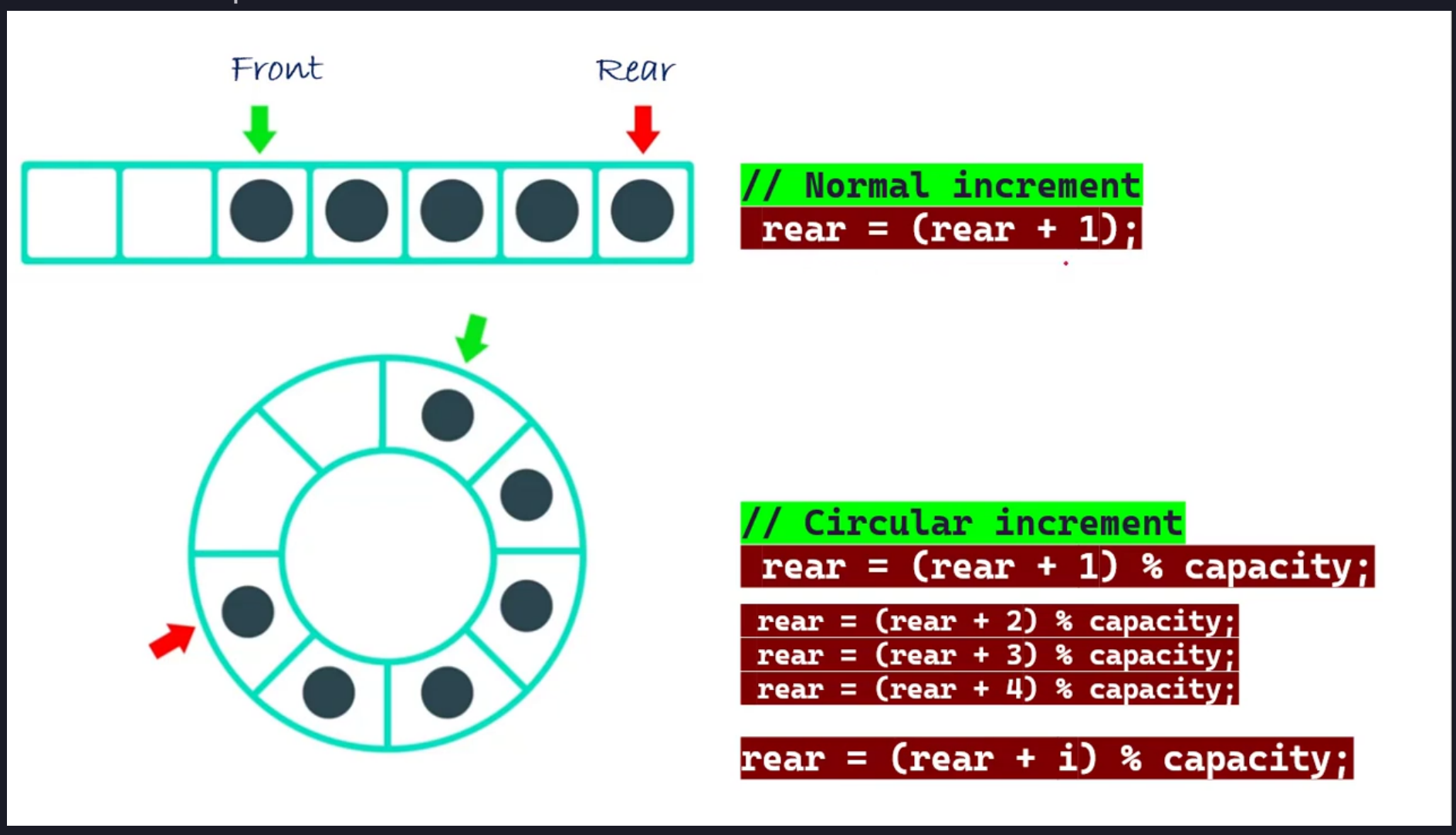
It is a linear-squential-logical data structure, follows the **First In, First Out (FIFO)** principle, meaning the first element added is the first one to be removed. This ordering is similar to people standing in a line: the first person in line is served first.

Queue Operations

1. ***Enqueue (push)***: Adds an element to the end (rear) of the queue.
2. ***Dequeue (pop)***: Removes and returns the element at the front of the queue.
3. ***Peek/Front***: Retrieves the element at the front without removing it.

Types of queues

- **Simple Queue**: Elements are inserted from one end and removed from the other, It uses a normal dynamic array, It's not very effeicient for memory.
- **Circular Queue**: It is a more efficient version of the simple queue. It uses a circular structure to utilize memory optimally, The last position is connected back to the first, when the rear reaches the end of the array and there's space at the beginning, it wraps around and uses this space.


Front
Rear
// Normal increment
rear = (rear + 1);
// Circular increment
rear = (rear + 1) % capacity;
rear = (rear + 2) % capacity;
rear = (rear + 3) % capacity;
rear = (rear + 4) % capacity;
rear = (rear + i) % capacity;

Ramez Medhat

- **Priority Queue**: Elements are removed and ordered based on priority, not based on the order they were pushed, It's implemented using heaps to allow for efficient priority-based retrieval.
- **Deque (double ended queue)**: A deque allows insertion and removal of elements from both the front and the rear of the queue. This flexibility makes it a handy tool for various applications where elements need to be processed from both ends.

Implementations

Queue can be built with both dynamic array or doubly linked list.

الاسهل تستخدم `linked list` كل اللي عليك فال `push` تضيف عنصر فالاخر وفال `pop` تمسح عنصر من *الاول* وفال `peek` هترجع العنصر اللي عند ال *head* (اللي فالاول)

Enqueue

```
void enqueue(t val){
    // allocation
    Node<t> *newNode = new Node<t>(val);

    if(head == nullptr) {
        head = newNode;
        tail = newNode;
    } else {
        tail→next = newNode;
        tail = newNode;
    }
}
```

Dequeue

```
void dequeue(){
    if(head == nullptr){
        cout << "Queue is empty!\n";
    } else {
        Node<t> *temp = head;
        head = head→next;
        delete temp;
    }
}
```

Top

```
void top(){
    if(head == nullptr){
        cout << "Queue is empty!\n";
    } else {
```

```
        cout << head→data << "\n";
    }
}
```

ولو عملته باستخدام دايناميك ارراي هتضيف عند ال size وتمسح من عند بداية الارراي

Priority Queue

A **priority queue** is a linear-squential-logical data structure similar to a queue, but each element is associated with a **priority**. Instead of following the **First In, First Out (FIFO)** principle, the element with the **highest priority** is dequeued first, regardless of the order in which elements were added. If two elements have the same priority, their order is determined by their insertion order or implementation-specific rules. This structure is like a VIP line where individuals with higher importance are served before others.

هتبقى زي الqueue العادية هنستخدم linked list بس لما نييجي ن insert هندور على العنصر اللي الpriority بتاعته اقل من العنصر ده على طول ونinsert بعده

Implementation

Node

```
template <class t>
struct Node {
    t data;
    int priority;  // priority of insert
    Node *next;

    Node(int val, int pri) : data(val), priority(pri), next(nullptr) {}
};
```

class

```
template <class t>
class PriorityQueue{
    Node<t> *head;
    int size;

public:
    PriorityQueue(){
        head = nullptr;
        size = 0;
    }
```

```
    ~PriorityQueue(){
        clear();
    }
};
```

Enqueue (Insertion)

```
void enqueue(t val, int pri){
        // allocation
        Node<t> *newNode = new Node<t>(val, pri);

        if(head == nullptr || newNode->priority > head->priority) {
                // insert at beginning
                newNode->next = head;
                head = newNode;
        } else {
                Node<t> *temp = head;
                while (temp->next != nullptr && newNode->priority <= temp->next-
>priority) {
                        temp = temp->next;
                }

                newNode->next = temp->next;
                temp->next = newNode;
        }
}
```

Deletion (Deletion)

Same as normal queue.

Top/Peek

Same as normal queue.

Tree

A tree is a *nonlinear hierarchical* data structure that consists of nodes connected by edges.

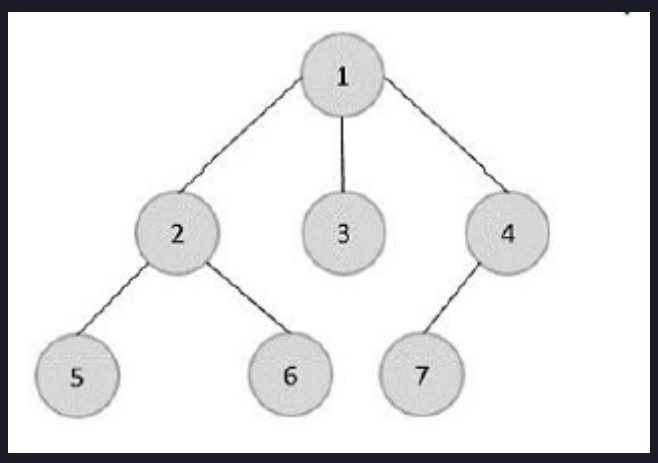
- **Node** - The fundamental part of a binary tree, where each node contains data and link to two child nodes.
- **Root** - The topmost node in a tree is known as the root node. It has no parent and serves as the starting point for all nodes in the tree.
- **Parent Node** - A node that has one or more child nodes.
- **Child Node** - A node that is a descendant of another node (its parent).

- **External nodes** (terminal nodes or Leaf Nodes) - A node that does not have any children or both children are null.
- **Internal Node** - A node that has at least one child. This includes all nodes except the root and the leaf nodes.
- **Edge** - A connection between two nodes. It represents the relationship between a parent and a child node.
- **Path** - A sequence of consecutive edges.
- **Subtree** - The Subtree is a portion of the tree that includes a node and all its descendants.

Types of trees

General tree

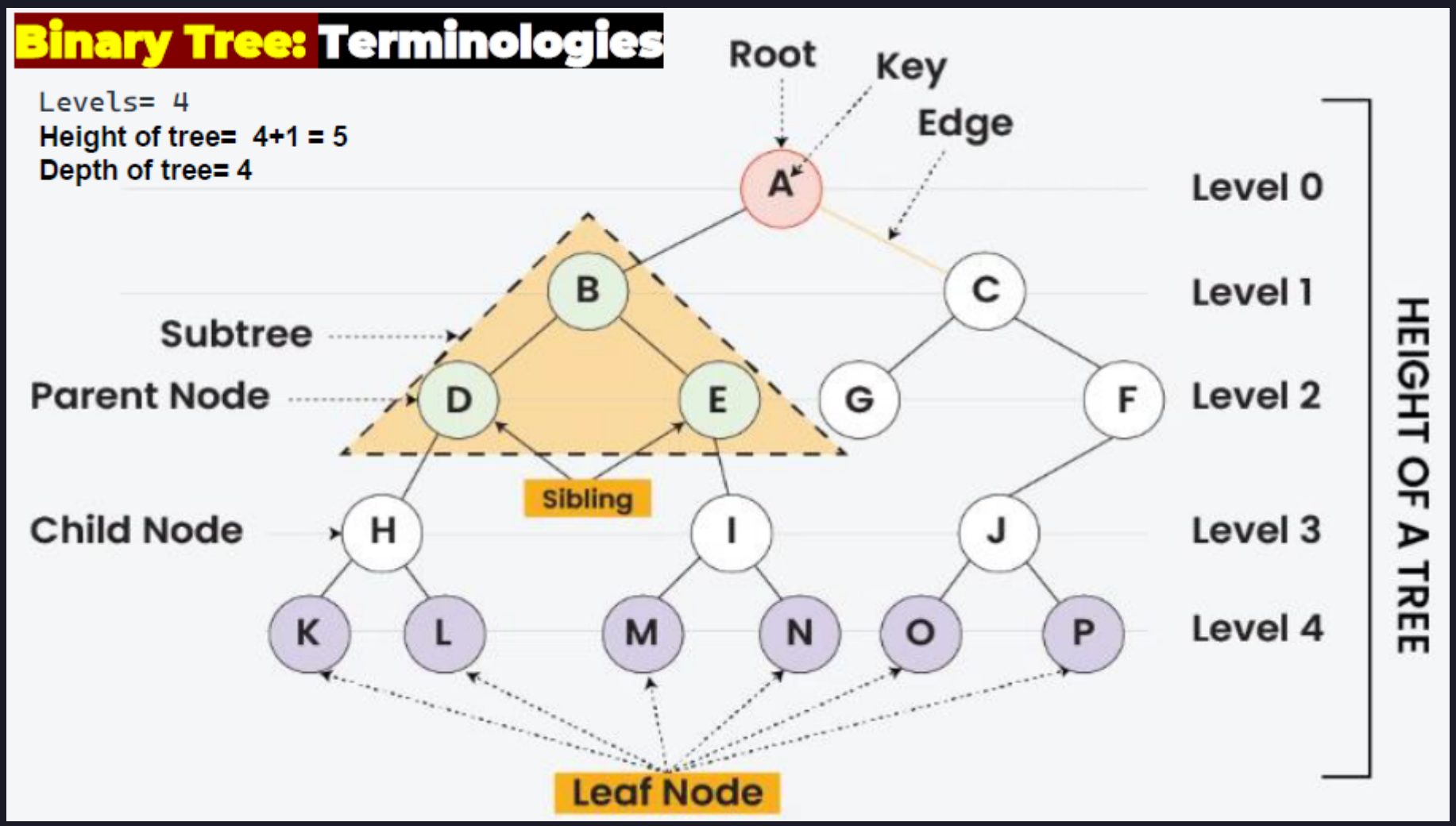
The root node has minimum 0 or maximum n subtrees. (No constrains on their hierarchy).



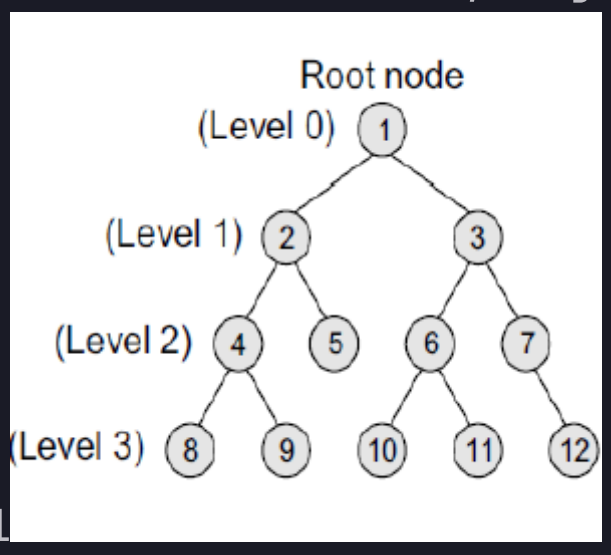
Binary tree

Each node has at most *two* children referred to as the left child and the right
child.
The *top most* node in a binary tree is called the *root*, and the *bottom most* nodes are called *leaves*.

Binary Tree Terminologies



- **Level** - The level of a node is defined by its depth. The root node is at level 0, its children are at level 1, and so The maximum number of nodes at any level L is 2^L.
- **Depth of a Node** - The number of edges length of the path from a specific node N to the root node R. The depth of the root node is zero. The maximum number of nodes in a binary tree of depth d is $2^d - 1$ where $(d > 1)$.
- **Height of a Binary Tree** - The total number of nodes on the path from the root node to the deepest leaf node in the tree. A tree with only a root node has a height of 1
- **Degree of a node** - It is equal to the number of children that a node has. The degree of a leaf node is zero For example, in the tree, degree of node 4 is 2, degree of node 5 is zero and degree of node 7 is 1



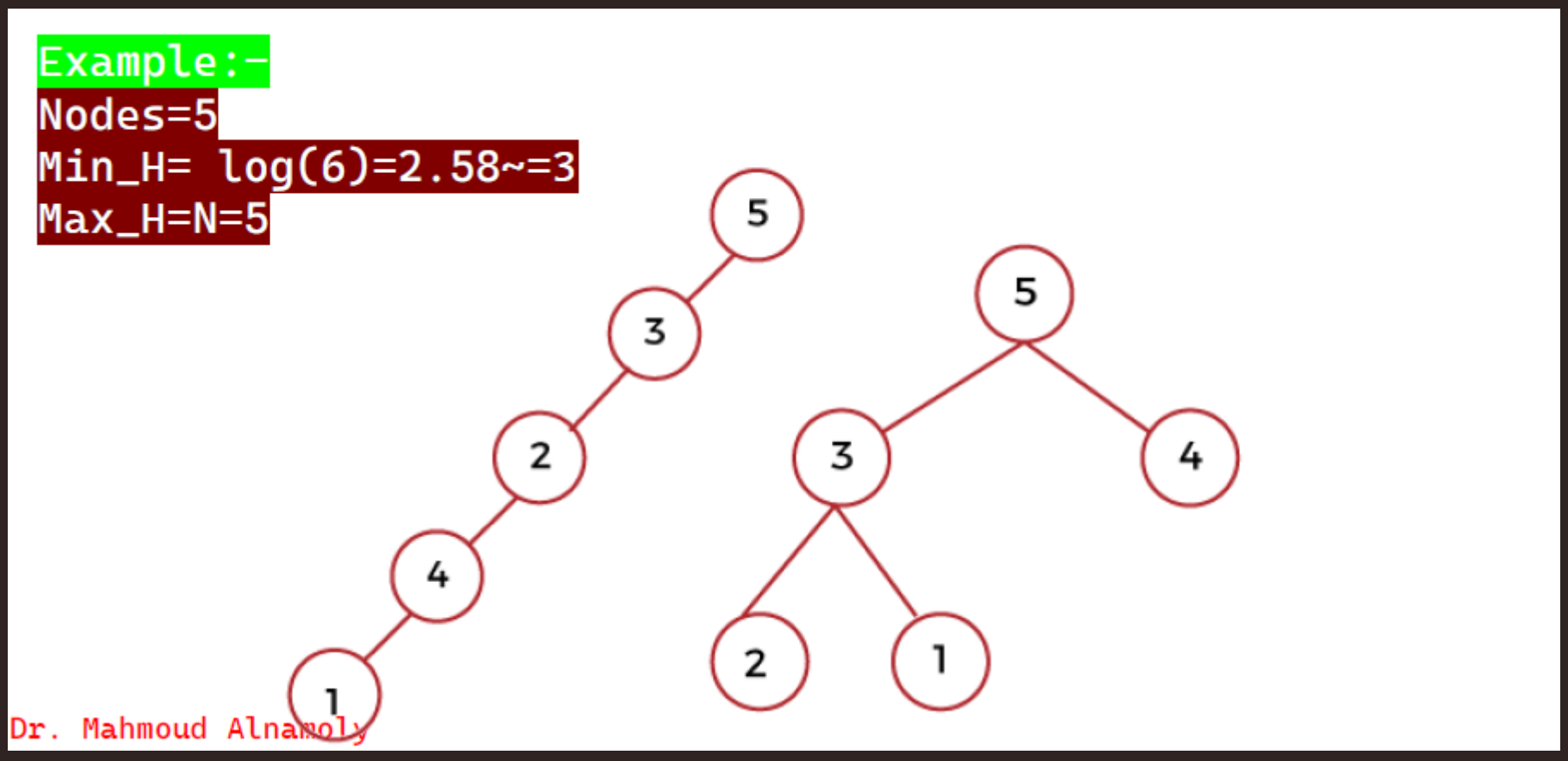
- **In degree/out degree of a node**
  - *In degree* is the number of edges arriving at a node. The root node is the only node that has an in degree equal to zero.
  - *Out degree* of a node is the number of edges leaving.

> ⚠ Note

The heigh of a binary tree with N nodes is

At Least: $ceil(log_2(N+1))$

At Most: N

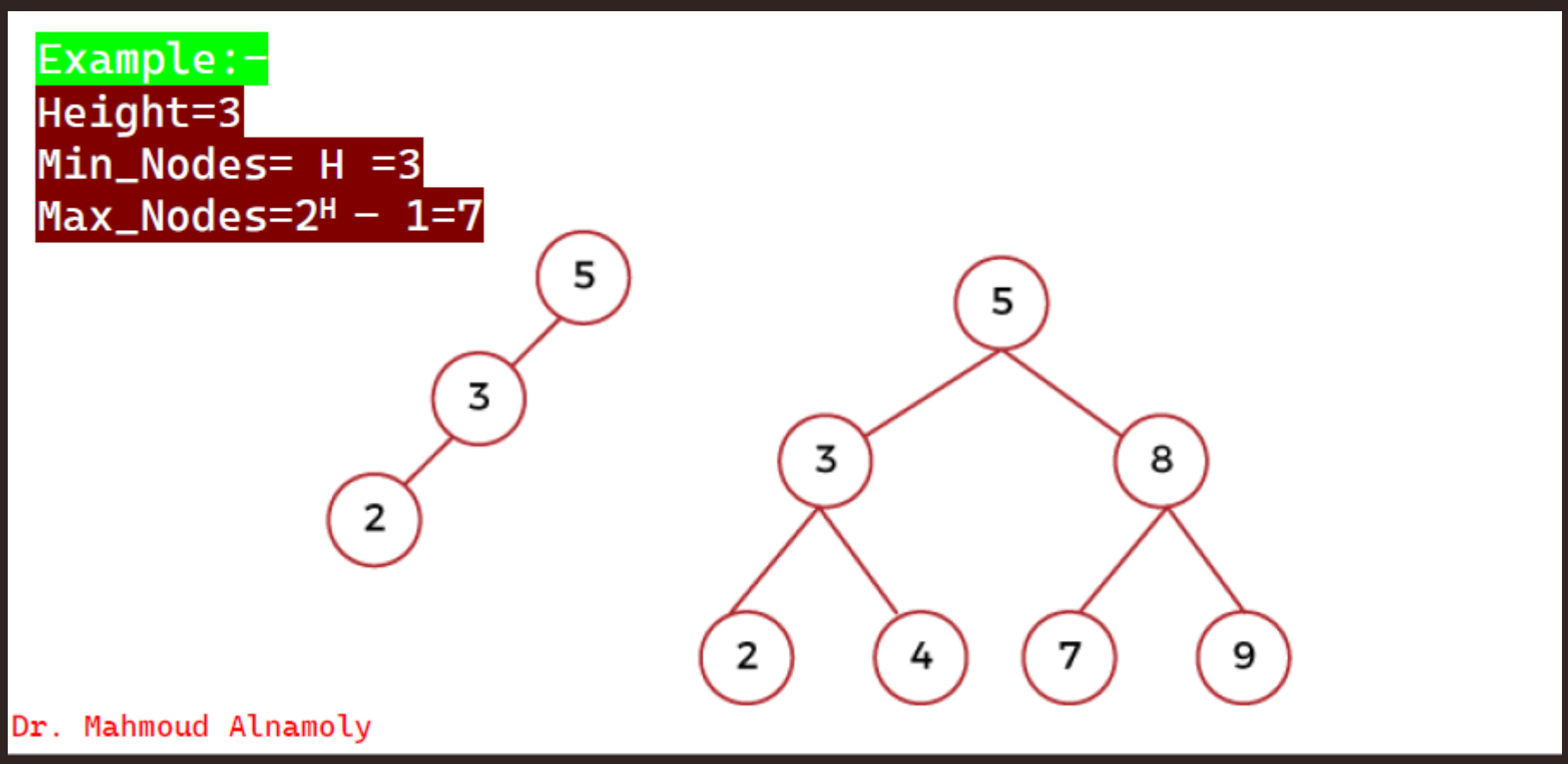


⚠ Note

The number of nodes on binary tree with height H is

At Least: H

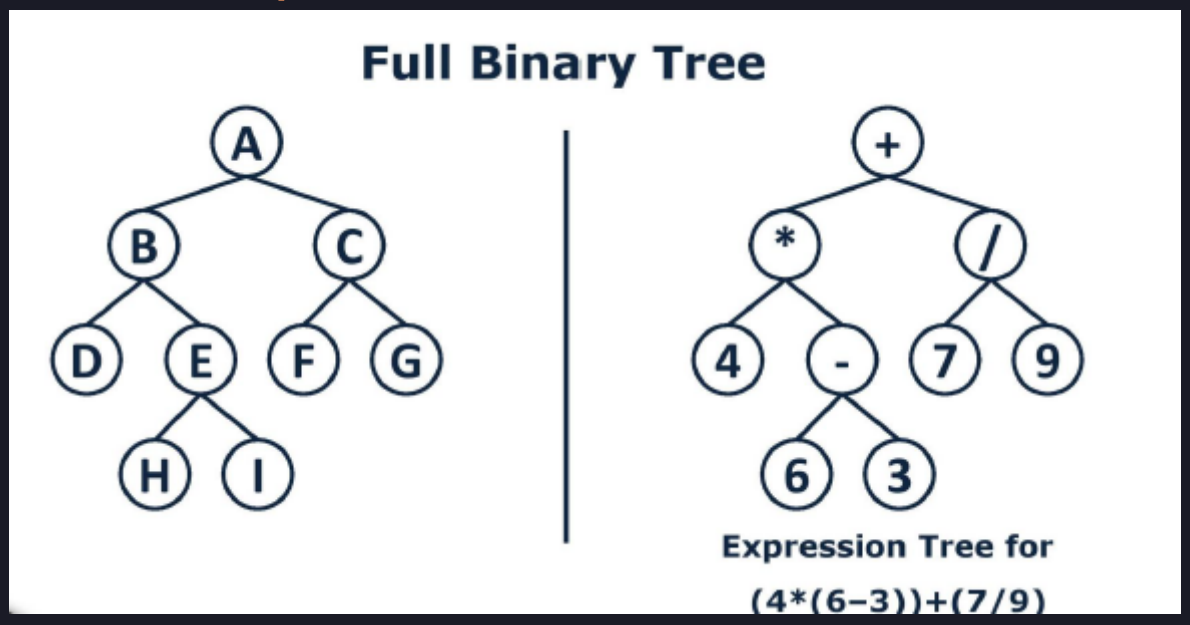
At Most: $2^H - 1$



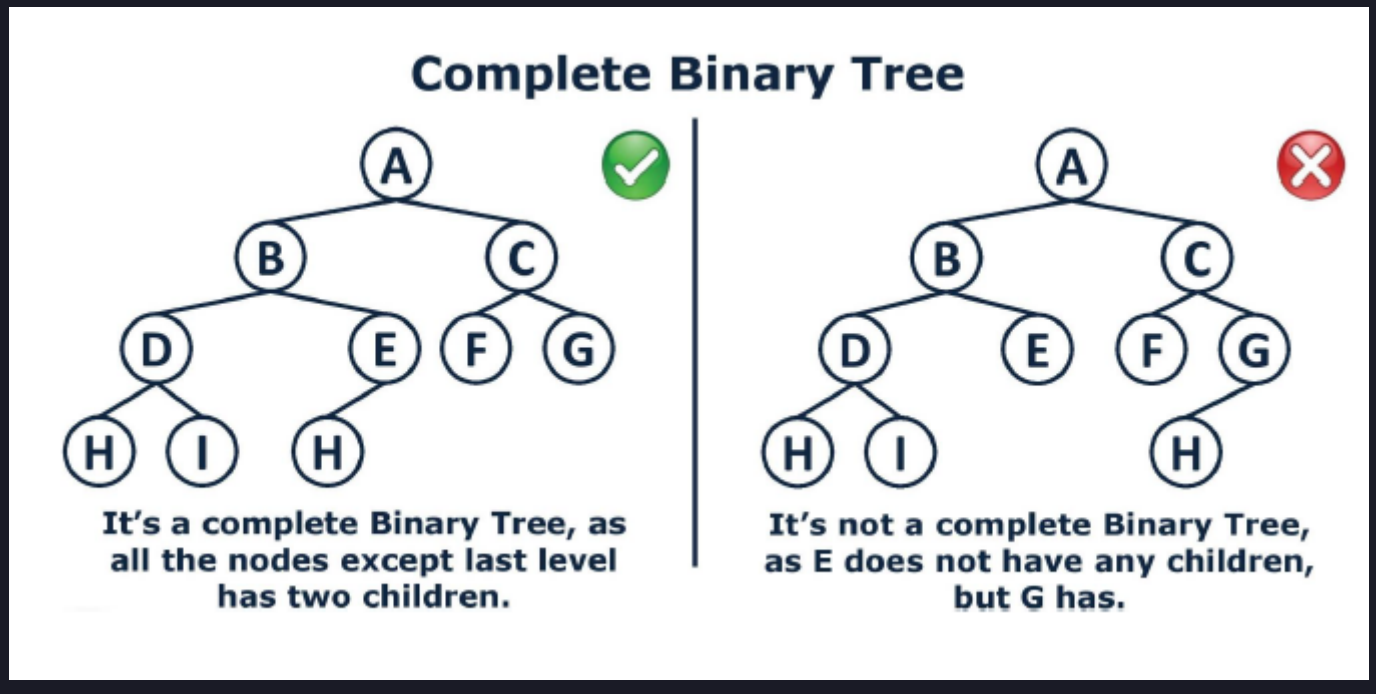
Binary Tree Variations

Ramez Medhat

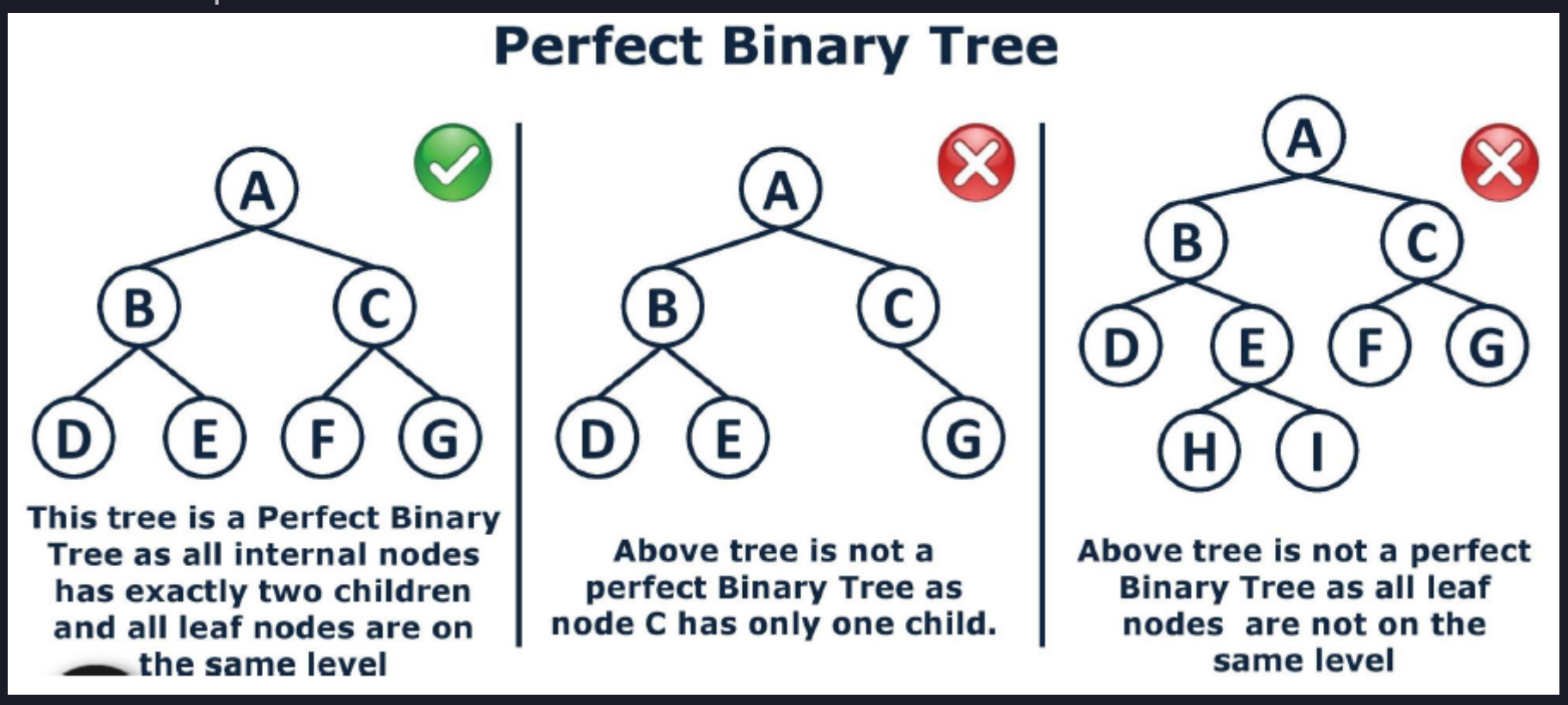
1. ***Full Binary Tree:*** each node has either *0 or 2* child node.



2. ***Complete binary tree:*** All levels, except possibly the last, are **completely filled** with nodes. Additionally, all nodes in the last level are positioned as far **left** as possible.



3. ***Perfect Binary Tree:*** all leaf nodes are on the same level, and every node except leaf nodes have 2 childern.



> Notes
>
> All perfect binary trees are complete, but reverese is not true.

Representation

Ramez Medhat

Node

Each node in a Binary tree has 3 parts

1. Data
2. Pointer to *Left Child*
3. Pointer to *Right Child*

```
struct Node{
    int data;
    Node* left;
    Node* right;
    Node(int value){
        data = value;
        left = right = nullptr;
    }
};
```

Tree

The tree it self is just a class holding a pointer to the root node.

```
class BinaryTree {
    Node* root;

public:
    BinaryTree() : root(nullptr) {}
    BinaryTree(Node *node) : root(node) {}
};
```

Traverse

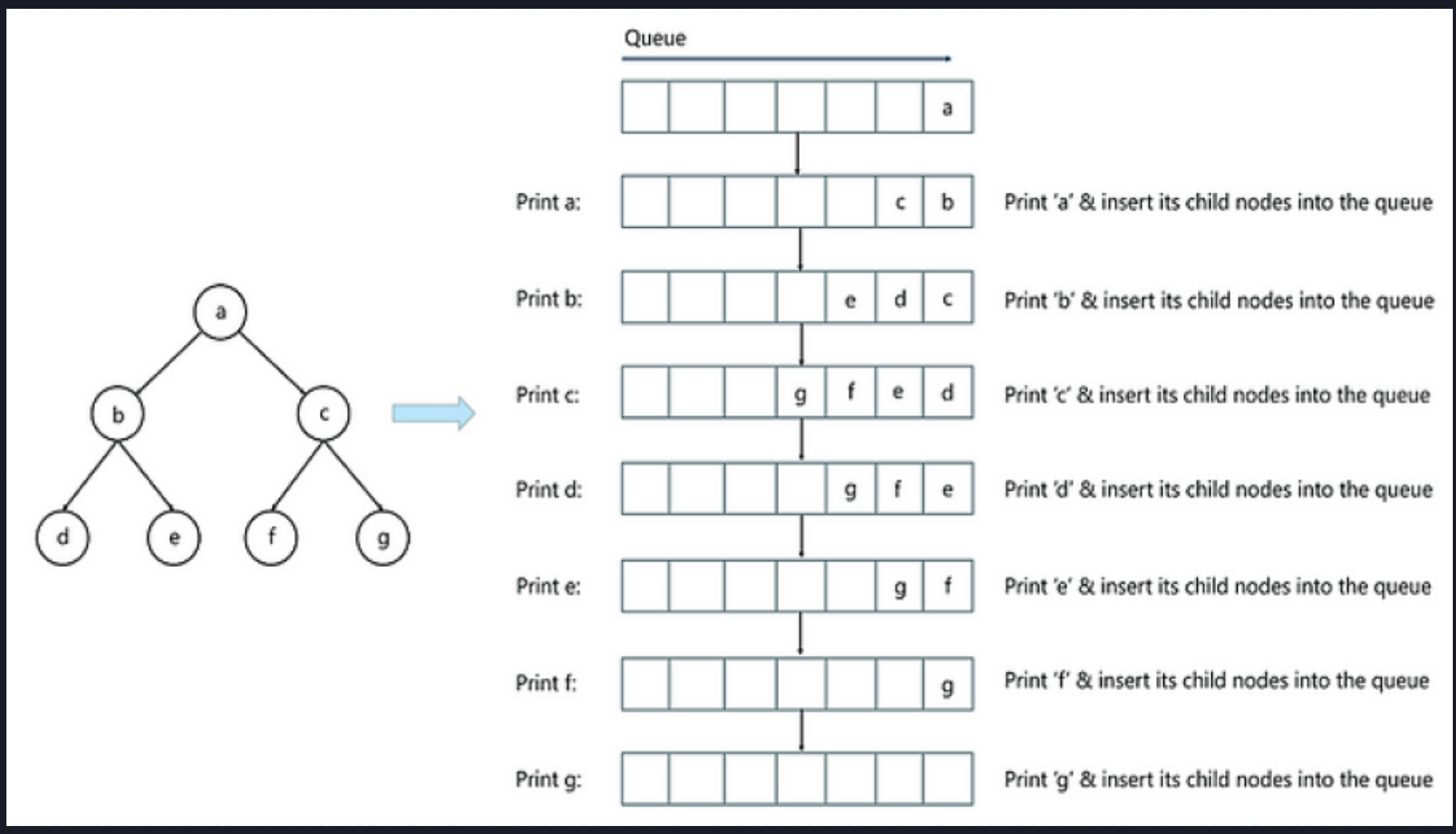
Visitng all nodes of the bianry tree using *BFS or DFS*.

Breadth First Search BFS algorithm

- BFS explores all nodes at the present depth before moving on to nodes at the next depth level.
- Implemented using *queue*.
- It follows the *level-order* traversal method, which means every level is traversed completely before moving to the next one.

```
void BFS(Node *root){
    if(root == nullptr) return;
    queue<Node*> q;
    q.push(root);
```

```
    while(!q.empty()){
        Node *node = q.front();
        q.pop();
        cout << node->data << " ";
        if(node->left != nullptr) q.push(node->left);
        if(node->right != nullptr) q.push(node->right);
    }
}
```



Depth First Search DFS Algorithm

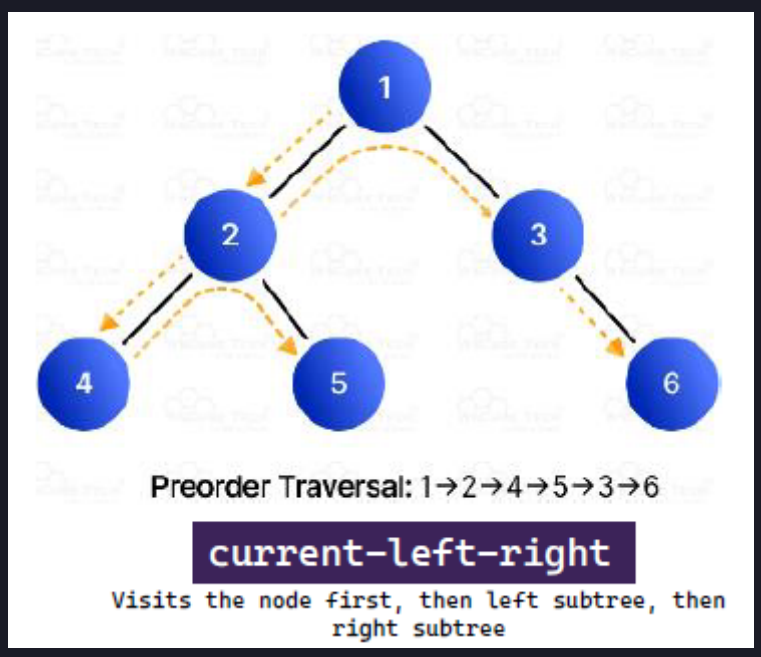
- DFS explores as far down a branch as possible before backtracking.
- It is implemented using *recursion*
- There are main traversal method in DFS which are (*Preorder - Inorder - Postorder*)

Preorder

Current → Left → Right.

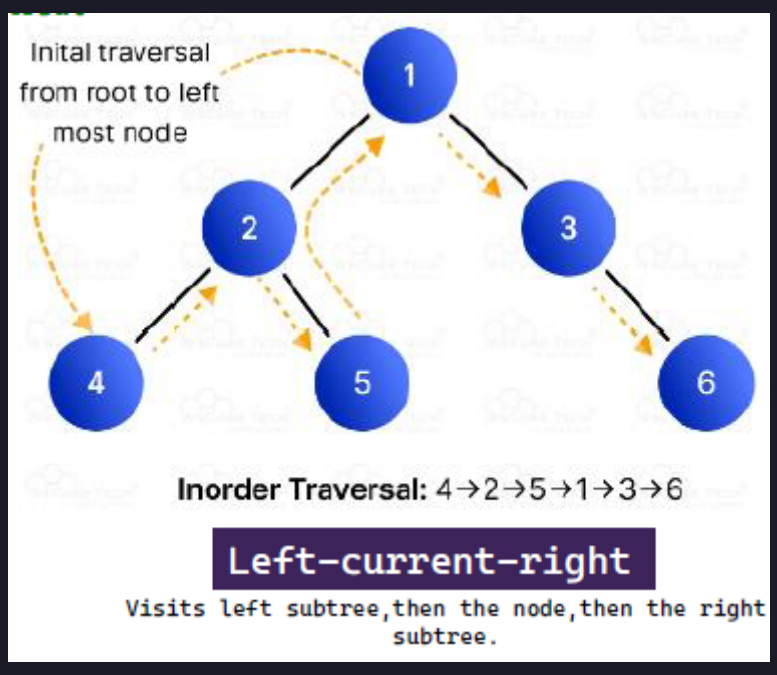
```
\\ helper function
void preorder(Node *node){
    if(node == nullptr){
        return;
    }

    cout << node->data << " ";
    preorder(node->left);
    preorder(node->right);
}

public:
```

```
void preorder(){
    preorder(root);
    cout << endl;
}
```



Inorder

Left → Current → Right.

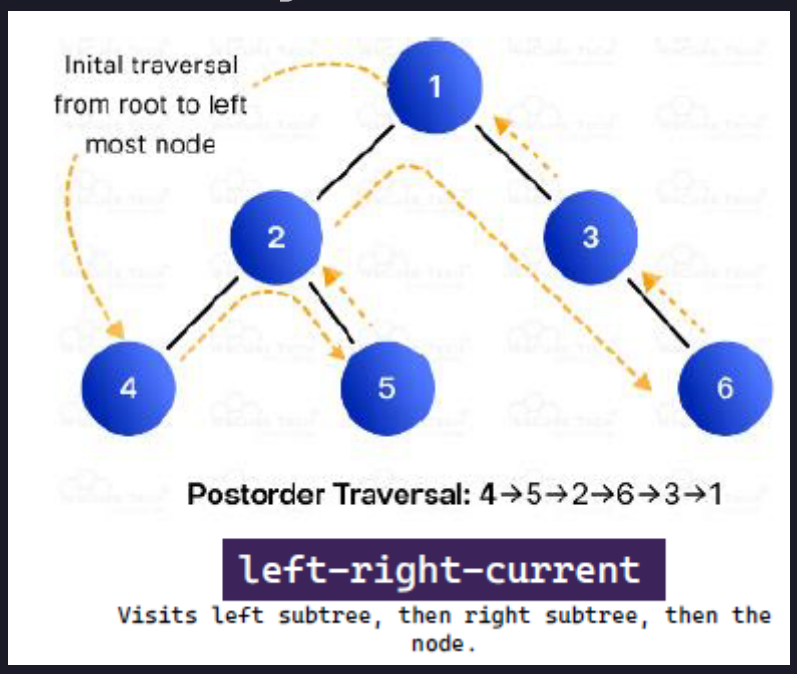



```
void inorder(Node *node){
    if(node == nullptr){
        return;
    }


    inorder(node->left);
    cout << node->data << " ";
    inorder(node->right);
}

public:
void inorder(){
    inorder(root);
    cout << endl;
}
```

Ramez Medhat

Postorder

Left → Right → Current.




```
void postorder(Node *node){
    if(node == nullptr){
        return;
    }


    postorder(node->left);
    postorder(node->right);
    cout << node->data << " ";
}

public:
void postorder(){
    postorder(root);
    cout << endl;
}
```

Insertion

We do not have to worry about the ordering of nodes in the binary tree.

1. We would first create *a root node* in case of *empty tree.*
2. Then *subsequent insertions* involve iteratively searching for *an empty place* at *each level of the tree*.
3. When an empty left or right child is found, then *a new node is inserted there.*
4. By convention, insertion always starts with the *left child node.*

```
void insert(int key){
    // if the tree is empty create the root
    if(root == nullptr){
        root = new Node(key);
    }
```

```
    // Create a queue for level order traversal (BFS)
    queue<Node*> q;
    q.push(root);

    while(!q.empty()){
        Node* temp = q.front();
        q.pop();

        // if left is empty insert at left else push the left in the queue
        if(temp->left == nullptr){
            temp->left = new Node(key);
            break;
        } else {
            q.push(temp->left);
        }

        // if right is empty insert at right else push the right in the queue
        if(temp->right == nullptr){
            temp->right = new Node(key);
            break;
        } else {
            q.push(temp->right);
        }
    }
}
```

Searching

We continue the prcesses of DFS or BFS process until we either *find the node* with the desired value or *reach the end of the tree.* If the tree is empty or the value isn't found after exploring *all possibilities*, we conclude that the value *does not exist in the tree.*

```
// helper function
bool searchDFS(Node* parent, int key){
    // if the current node is null then we reached the leaf
    if (parent == nullptr) return false;

    // If the node's data is equal to the value we are searching for
    if (parent->data == key) return true;

    // Recursively search in the left and right subtrees
    bool left_res = searchDFS(parent->left, key);
    bool right_res = searchDFS(parent->right, key);

    return left_res || right_res;
}

bool searchDFS(int key){
```

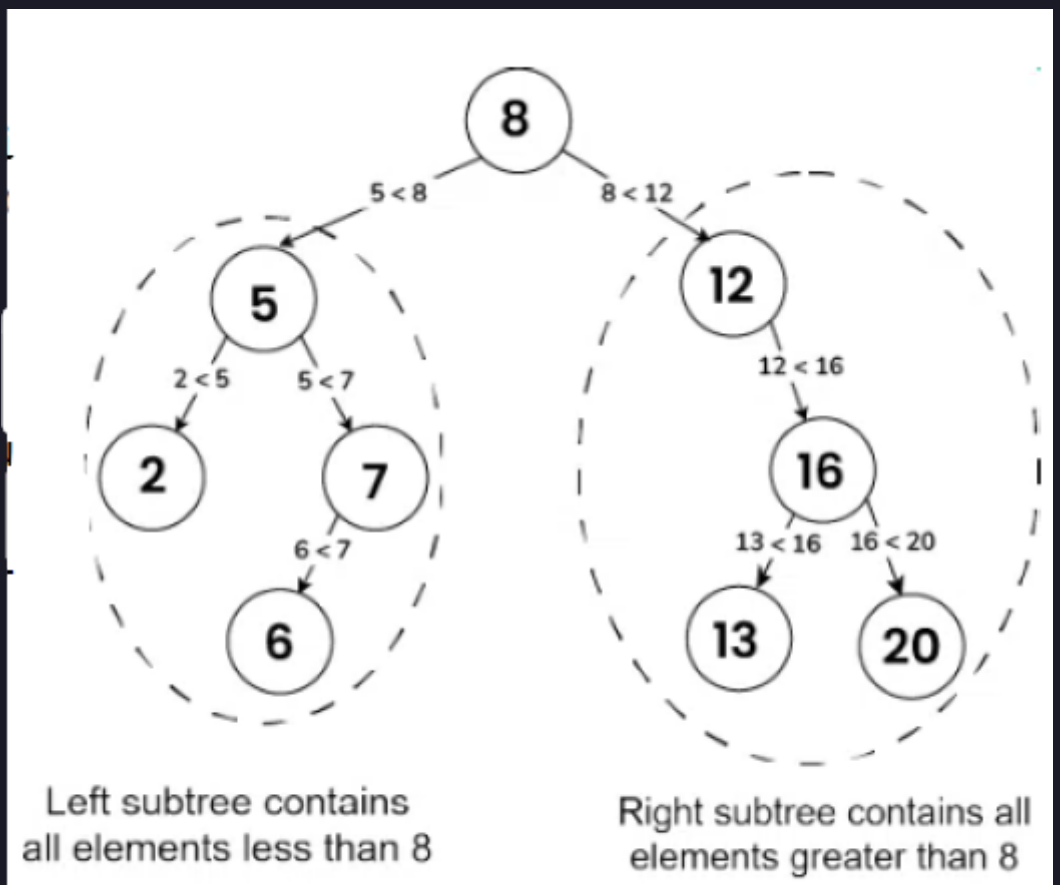
```
    return searchDFS(root, key);
}
```

Complexity of code as n is the number of nodes

Operation	Time Complexity	Auxiliary Space
In-Order Traversal	O(n)	O(n)
Pre-Order Traversal	O(n)	O(n)
Post-Order Traversal	O(n)	O(n)
Insertion (Unbalanced)	O(n)	O(n)
Searching (Unbalanced)	O(n)	O(n)
Deletion (Unbalanced)	O(n)	O(n)

Binary Search Tree BST

- It possess all the properties of Binary Trees including some extra properties
- The values in **the left subtree** are always **less** than the values in **the root** node and the values in **the right subtree** are always **greater** than the values in **the root** node
- This hierarchical structure allows for ***efficient*** Searching, Insertion, and Deletion operations on the data stored in the tree.



`left subtree<root node≤ right subtree`

BST Operations

Insertion

- Start at the root.
- Compare the value to be inserted with the current node's value:
  1. If `key < node value`: Move to the left subtree.
  2. If `key > node value`: Move to the right subtree.
- When you find an empty spot (null), insert the new node there.

```
// helper function
void insert(Node* node ,int key){
    // If the tree is empty, create the root
    if (root == nullptr) {
        root = new Node(key);
        return;
    }

    // BST allows no repetition, so if the value already exists, we do nothing
    if (key == node->data) {
        return;
    } else if (key < node->data) {
        // If left child is null, create a new node
        if (node->left == nullptr) {
            node->left = new Node(key);
        } else {
            // Recur for the left subtree
            insert(node->left, key);
        }
    } else { // key > node->data
        // If right child is null, create a new node        if (node->right ==
```

```
nullptr) {
            node->right = new Node(key);
        } else {
            // Recur for the right subtree
            insert(node->right, key);
        }
    }
}

void insert(int key){
    insert(root, key);
}
```

Search

- Start at the root.
- Compare the value to be searched (let's call it `key`) with the current node's value:
  1. If `key == node value`: The key is found.
  2. If `key < node value`: Recursively search the left subtree.
  3. If `key > node value`: Recursively search the right subtree.
- If you reach a null pointer (leaf node), the key is not in the tree.

```
// Helper function
bool search(Node* node, int key) {
    // Base case: If the node is null, the key is not found
    if (node == nullptr) {
        return false;
    }

    // If the key matches the current node's data, it is found
    if (key == node->data) {
        return true;
    }

    // Recur to the left subtree if the key is smaller
    if (key < node->data) {
        return search(node->left, key);
    } else { // Recur to the right subtree if the key is larger
        return search(node->right, key);
    }
}

bool search(int key){
    return search(root, key);
}
```

Advantages of BST

Binary Search Trees are **more efficient** than binary trees, insertion and search is done in complexity of `O(log n)` where n is the number of nodes.

Disadvantages

The main disadvantage of Binary Search Trees is that if all elements in nodes are either greater than or lesser than the root node, the tree becomes ***skewed***. Simply, the tree becomes ***slanted to one side completely.***

This skewness will make the tree a linked list rather than a BST, since the worst case time complexity for searching operation becomes `O(n)`

To overcome this issue of skewness in the Binary Search Trees, *the concept of Balanced Binary Search Trees* was introduced.